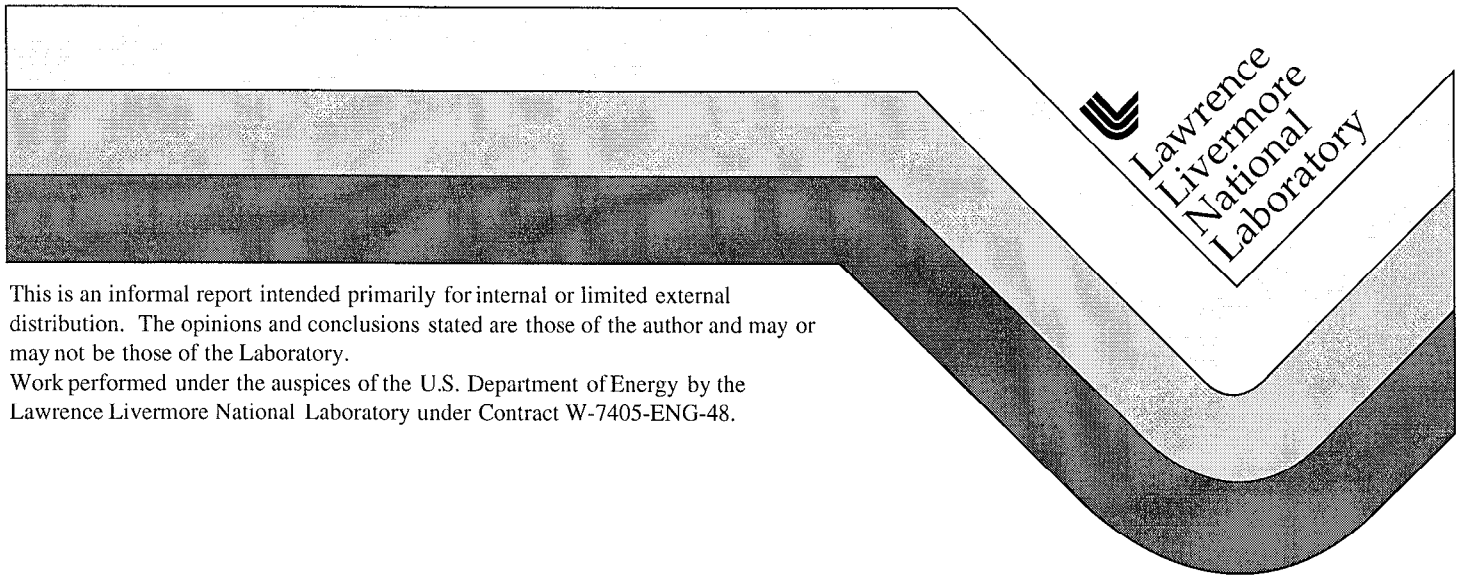
UCRL-ID-135342

Solvers as Operators

M.G. Knepley
A.J. Cleary

August 2, 1999



This is an informal report intended primarily for internal or limited external distribution. The opinions and conclusions stated are those of the author and may or may not be those of the Laboratory.
Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract W-7405-ENG-48.

DISCLAIMER

This document was prepared as an acccount of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately own rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This report has been reproduced
directly from the best available copy.

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information
P.O. Box 62, Oak Ridge, TN 37831
Prices available from (615) 576-8401, FTS 626-8401

Available to the public from the
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd.,
Springfield, VA 22161

Solvers as Operators

Proposal for the ESI Solver Interface

Matthew G. Knepley
Computer Science Department
Purdue University
West Lafayette, IN 47906-1398
knepley@cs.purdue.edu
Phone: (765) 494 7816, FAX: (765) 494 0739

Andrew J. Cleary
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94550
cleary1@llnl.gov
Phone: (925) 422 1939, FAX: (925) 423 2993

This proposal details an interface hierarchy for objects whose major function is the mapping of a finite dimensional vector space of dimension m to another vector space of dimension n. This includes many important objects in a solver library, including matrices, their transposes and inverses, solvers, preconditioners, iterative methods, and nonlinear maps. A unifying framework for finite dimensional operators and solvers is proposed which utilizes the composition operation from the operator algebra to achieve great functionality while reducing the size of the interface and complexity of the class structure. A second composition operation is introduced to handle the composition of approximate solution techniques, and related to several common preconditioning techniques.

Solvers as Operators

The goal of the model presented here is to derive a common abstraction from these objects, incorporating the matheamtical structure of these maps into a single interface, so that clients of these objects can use them in a uniform manner, and to introduce operations on these objects that define an algebra that provides for a powerful mechanism for generating new operators from old operators in a flexible and extensible way. Some benefits of general algebraic constructions have already been identified, but perhaps more importantly, this rich structure leaves room for combinations that may be developed by future research, thus greatly increasing the chances that the ESI standard will be sufficient for expressing future generations of algorithms. Standard efforts frequently suffer from rapid obsolescence, and a standard that can grow with time is highly desirable.

At the same time, it is essential that making legacy code ESI compliant not impose an undue burden on the original programmer. To this end, we show how solvers written in several styles can be made compliant with this proposal in a straightforward way. In some cases there are several

options for achieving compliance, which is consistent with the ESI strategy of putting the burden of choice on developers writing to the ESI standard rather than on the crafters of the standard.

The most controversial aspect of this proposal is that it does not include a specific preconditioner class. It is our stance that preconditioning is a *role* played by specific operators in certain circumstances, rather than a separate type of object meriting its own class. As a simple analogy, consider a banking system which includes a multipurpose Customer class. When implementing a Transaction interface, it is desirable to have an associated transaction_customer member. The implementers may choose to make transaction_customer a Customer, or to develop a more specific TransactionCustomer class. If the first choice is made, no new class is needed, and objects of type Customer can be transparently exchanged between Transactions and other parts of the system. On the other hand, if requiring a transaction_customer to be of type Customer introduces significantly new requirements for the Customer class, then the second solution may make more sense, particularly if TransactionCustomer is a subclass of Customer. For preconditioners we have a similar choice, and we argue that there is an enormous utility in retaining the generality of the Operator interface for both solvers and preconditioners.

The competing proposal makes the argument that preconditioning can take several forms requiring different things from the preconditioner, and thus a utility class is a better solution. However, we argue in this proposal that the extra functionality required of special preconditioners, which include split preconditioners and *efficiency–trick* preconditioners like Eisenstat's SSOR, may be derived in a straightforward way from the underlying mathematical structure of the space of operators embodied in our interface. Thus, while we believe that both proposals address the extant scenarios that ESI has put forth, our proposal does so in a way that is more flexible in that it does not hard code the preconditioning relationship into a class. Our model also allows ESI–compliant classes to be easily combined in unanticipated ways without changing the interface. Consider a user who would like to precondition a solver with a matrix that directly approximates the inverse of the system operator. In our model, this matrix could be used directly as a preconditioner without constructing a wrapper. In the previous model, only an object of type Preconditioner can be used as a preconditioner to a solver, so the user must write or borrow code that can encapsulate their matrix as a Preconditioner. We view easy plug–and–play behavior at the user level as extremely important, and requiring even straightforward wrapper code a significant hurdle, enough to discourage some users from experimenting with novel solution strategies. Also, writing ESI–compliant code may be subtly harder than one would expect. For instance, it is not clear which language the user should choose for this wrapper. While the same question is valid for developers of ESI compliant components, we prefer to put the burden of implementation on knowledgable developers, where the cost is amortized over all users of a particular component, rather than on users, where the cost is multiplied by the number of users.

Similarly, this proposal includes in the interface an algebra for approximate solution techniques. This algebra is based upon subspace or residual correction, a general concept important in the theory of iterative methods. Again, while the functionality of this approach can be coded directly by developers, by defining the algebra, the model allows users to experiment with various combinations of iterative methods using any appropriate ESI–compliant objects, without writing new wrapper classes or auxilliary logic. The necessary operations are written once by developers and that cost is amortized over all user uses.

2. Interface Hierarchy

■ Operators

The motivation for establishing the **Operator** interface is to abstract out the common features of arbitrary nonlinear finite dimensional maps, which may correspond to both exact and approximate representations of infinite dimensional maps in some Banach space, as well as arbitrary finite dimensional transformations. Thus the **Operator** interface must contain an **apply** function which takes as input an **ESI_Vector** and returns the action of that operator as output in another **ESI_Vector**. The necessity of supporting Fortran may prohibit aliasing of the input vector to the output vector. The **Operator** interface will also contain a **setup** method so that any auxiliary data structures or calculation may be handled prior to application, and a **getDimensions** method returning the sizes of the domain and range spaces.

The application of the transpose and hermitian conjugate are realized by constructing a new operator from an existing one, whose **apply** function implements that operation. The **OperatorTranspose** interface, for instance, would be implemented by those operators able to construct such a transpose. Reference counting techniques should allow data structures to be efficiently shared between the instantiations. This is also the philosophy we will use in order to derive the **Solver** interfaces, and to deal with preconditioner abstractions. Finally, an algebra (and in special cases group structure) can be derived for the **Operator** interface so that new **Operator** objects can be constructed through algebraic operations such as composition (or inversion in a group). The aspects of this composition property will be discussed further in sections 4 and 5.

This encapsulation of various actions of the operator in separate objects is an instance of the **Strategy** pattern[2]. Abstractly, this pattern defines a family of algorithms, each one of which is encapsulated in a separate object, which all implement a uniform interface. In this way the algorithm can vary independently of the client code which manipulates it. The **Operator** interface functions as the **Compositor** in the pattern with the uniform interface **apply** corrseponding to **Compose**. This also allows the tranpose or conjugate to be passed independently to other algorithms without modification.

```
interface ESI_Operator extends ESI_Object {
  int setup();
  int getDimensions(out int m, out int n);
  int apply(in ESI_Vector x, out ESI_Vector y);

  int composeLeft(in ESI_Operator G, out ESI_Operator GF);
  int composeRight(in ESI_Operator G, out ESI_Operator FG);
}
```

```
interface ESI_OperatorTranspose extends ESI_Operator {
  int createTranspose(out ESI_Operator FT);
}

interface ESI_OperatorHermitian extends ESI_Operator {
  int createHermitian(out ESI_Operator FH);
}
```

The **OperatorSplittable** interface allows for easily decomposable operators, such as Cholesky factorizations, to export this functionality. Again, we encapsulate the new action in an object, so that we can leverage our entire algebraic apparatus with that object as well. Notice that the **composeSplit** function allows for sophisticated optimization, such as the introduction of Eisenstat's Trick, since the entire state of the operator is available to the implementor.

```
interface ESI_OperatorSplittable extends ESI_Operator {
  int split(out ESI_Operator L, out ESI_Operator R);
  int composeSplit(in ESI_Operator F, out ESI_Operator LFR);
}
```

■ Matrices

The **Matrix** interface encapsulates the behavior of a linear operator. Thus it does not impact the **apply** function, but the assumption of linearity will be useful when using the algebraic operations on operators. Also, information about the eigenstructure could be included in this interface. The interfaces for deriving transposes and conjugates might also be replicated.

```
interface ESI_Matrix extends ESI_Operator {
}
```

■ Solvers

The **Solver** interface is meant to abstract the action of the inverse of some operator, whether exact or approximate. The **getOperator** and **setOperator** functions provide access to the designated operator, and the **apply** function now gives gives the action of the inverse.

Any **Solver** may also be provided a preconditioner in order to facilitate calculation in the **apply** function. The preconditioner is an **Operator** which may be applied in the course of the computation. It could be argued that the preconditioner M should be a **Matrix**, which would include the case of nested preconditioning using linear solvers, but not nonlinear ones. Notice that we do not require a separate interface for preconditioners since the traditional logic may be expressed in terms of the algebra of operators. Examples of traditional preconditioning approaches will be shown in section 3 to demonstrate the efficacy of this design.

```
interface ESI_Solver extends ESI_Operator {
  int getOperator(out ESI_Operator F);
  int setOperator(in  ESI_Operator F);

  int getPreconditioner(out ESI_Operator G);
  int setPreconditioner(in  ESI_Operator G);

  int getResidual(out ESI_Vector r);
  int setResidual(in  ESI_Vector r);

  int resComposeLeft(in ESI_Solver T, out ESI_Solver TS);
  int resComposeRight(in ESI_Solver T, out ESI_Solver ST);
}
```

A straightforward extension of the discussion for the **Operator** interface argues that linear solvers should extend the **Matrix** interface since that inverse would be part of the same group GL(N).

```
interface ESI_LinearSolver extends ESI_Matrix, ESI_Solver {
  int getMatrix(out ESI_Matrix A);
  int setMatrix(in  ESI_Matrix A);
}
```

The **SolverIterative** interface makes explicit the assumption that the action of inverse is approximate and that the approximation is controlled by a **ConvergenceTest** object. We have replaced the usual scalar tolerance and maximum number of iterations with an interface to allow more general, user–specified stopping criteria.

```
interface ESI_SolverIterative extends ESI_Solver {
  int getNumIterations(out int numIterations);
  int setNumIterations(in  int numIterations);

  int getConvergenceTest(out ESI_ConvergenceTest test);
  int setConvergenceTest(in  ESI_ConvergenceTest test);
}
```

```
interface ESI_ConvergenceTest extends ESI_Object {
  int isConverged(in ESI_SolverIterative solver, out BOOL converged);
}
```

3. Example Implementations

■ Gaussian Elimination

Direct solvers are much more straightforward from the perspective of solver design, and thus we will use Gaussian elimination to illustrate the design goals of our approach. Two concrete solvers are introduced, **GaussianEliminator** and **SolverTriangular**, both of which are direct solvers derived from **ESI_SolverLinear**. Only the implementation of **GaussianEliminator** is shown as **SolverTriangular** is straightforward. **GaussianEliminator** also inherits from **ESI_OperatorSplittable** in order to demonstrate the functionality of that interface. The complete implementation of this example may be found in the ESI Forum[1].

The **setup** function constructs the decomposition of `A` and stores `L` and `U` as **ESI_Matrix** objects. In addition, it creates two **SolverTriangular** objects which are passed `L` and `U` which encapsulate the backward and forward solves. Then in **apply**, we merely execute the **apply** functions of these triangular solvers. In this way we avoid a messy **solve** function in the **ESI_Operator** interface, and also enable composition using the inverses.

```
class GaussianEliminator implements
ESI_SolverLinear,ESI_OperatorSplittable {
  ESI_Matrix L,U;
  ESI_Solver invL,invU;
  int        setupState;
}
```

```
int GaussianEliminator_setup()
{
  ESI_Matrix A;

  /* Check for a valid matrix to factor */
  getMatrix(&A);
  if (A == NULL) {
    return -1;
  }
  /*
    ESI compliant factorization code:
      Checks for acceptable matrix interface types
      Creates L and U based on A
  */
  SolverTriangular_create(invL);
  SolverTriangular_create(invU);
  SolverTriangular_setMatrix(invL, L);
  SolverTriangular_setMatrix(invU, U);
  /* Announce that the matrix was factored */
  setupState = 1;
  return 0;
}
```

```
int GaussianEliminator_apply(ESI_Vector x, ESI_Vector y) {
  ESI_Vector z;

  ESI_Vector_clone(x, &z);
  SolverTriangular_apply(invL, x, z);
  SolverTriangular_apply(invU, z, y);
  ESI_Vector_destroy(z);
  return 0;
}
```

The operator composition functions for this example are detailed in section. Those for split composition, however, are overridden by the solver, and are shown below.

```
int GaussianEliminator_split(ESI_Operator *L, ESI_Operator *R)
{
  *L = invU;
  *R = invL;
  return 0;
}
```

```
int GaussianEliminator_composeSplit(ESI_Operator A, ESI_Operator
*LAR)
{
  ESI_Operator_create(LAR);
  (*LAR)->composeList    =     ESI_Operator[3];
  (*LAR)->composeList[0] = invL;
  (*LAR)->composeList[1] = A;
  (*LAR)->composeList[2] = invU;
  return 0;
}
```

■ Incorporating legacy solvers

We would like to enable the user to wrap a legacy solver code so that it conforms to the ESI interface specification without impacting the functionality of the solver, and perhaps extending it. The next sections present three scenarios focusing on legacy Krylov solvers which involve different preconditioner interfaces. We demonstrate that each one can be incorporated into the ESI framework by embedding some logic into the wrapper code. This approach could be generalized by deriving a subinterface of **Solver** corresponding to each scenario.

▪ Solvers with explicit preconditioners

A solver which accepts a preconditioner may just utilize the **getPreconditioner** function in the **Solver** interface to retrieve the operator, which may then be applied using the **apply** function. For a legacy solver, this logic would reside in the wrapper code. As an example, we wrap a Petsc SLES object[4].

```
class PetscSLES implements ESI_SolverLinear {
  SLES sles;
}
```

```
int setup()
{
  ESI_Matrix    A;
  ESI_Operator M;

  getMatrix(&A);
  getPreconditioner(&M);

  SLESSetOperators(sles, A, M, DIFFERENT_NONZERO_PATTERN);
  return 0;
}

int apply(ESI_Vector x, ESI_Vector y)
{
  int its;

  SLESSolve(sles, x, y, &its);
  return 0;
}
```

▪ Solvers with explicit split preconditioners

A solver which expects a split preconditioner may again retrieve the operator using **getPreconditioner**, and then perform an interface query to check that it implements **OperatorSplittable**, which will be explained in detail in section 4. The factorization may then be retrieved using **split**, which again would reside in the wrapper code for legacy solvers. Thus we might have an implementation such as

```
class PetscSplitSLES implements ESI_SolverLinear {
  SLES sles;
}
```

```
int setup()
{
  ESI_Interface I;
  ESI_Matrix    A;
  ESI_Operator  M, L, R;

  getMatrix(&A);
  getPreconditioner(&M);
  M.queryInterface("ESI_OperatorSplittable", &I)
    (I == NULL)
      -1;
  M.split(&L, &R);

  SLESSetOperators(sles, A, L, R, DIFFERENT_NONZERO_PATTERN);
    0;
}
```

- **Solvers without explicit preconditioners**

A legacy solver which only requires the application of the system matrix may be wrapped in a straightforward manner using the **setup** and **solve** interface functions. In order to accomodate preconditioning, a new operator is derived by composition which is then passed to **solve**, and additional logic manages the input and output vectors.

```
class PetscSimpleSLES implements ESI_SolverLinear {
  SLES sles;
}
```

```
int setup()
{
  ESI_Interface I;
  ESI_Matrix    A, B;
  ESI_Operator  M;

  getMatrix(&A);
  getPreconditioner(&M);
  M.queryInterface("ESI_OperatorSplittable", &I)
    (I == NULL) {
    M.composeLeft(A, &B);
  }   {
    M.composeSplit(A, &B);
  }

  SLESSetOperator(sles, B, DIFFERENT_NONZERO_PATTERN);
    0;
}

int apply(ESI_Vector x, ESI_Vector y)
{
  ESI_Operator  M, L, R;
  ESI_Vector    r, s, t;
  int           its;

  getPreconditioner(&M);
  M.queryInterface("ESI_OperatorSplittable", &I)
    (I != NULL) {
    M.split(M, &L, &R);
  }

  /* Set up the shifted preconditioned system Bt = s with t_0 = 0 */
  A.apply(y, r);
  r.aypx(-1.0, x)
    (L != NULL) {
    L.apply(r, s);
  }   {
    M.apply(r, s);
  }
```

```
  SLESSolve(sles, s, t, &its);

  /* Preconditioned solution is in t.
     Recover the unpreconditioned solution, then shift it back by y */
     (R != NULL) {
    R.apply(t, r);
  }
  y.axpy(1.0, r);

       0;
}
```

4. Operator Composition

The algebraic operations on **Operators** take as input another **Operator** and produce a third which encapsulates the action of both. This designs mirrors the algebraic structure of finite–dimensional operators, and is also an instance of the **Composite** design pattern. This allows user code to treat individual operators and composed oeprators uniformly. The Composite pattern must have an abstract class the represents both the primitives and containers, which is naturally **Operator** since algebraically compositions are again operators. This greatly simplifies client code, which would normally have to write tag–and–case–statement–style functions to deal with composition.

```
int ESI_Operator_apply(ESI_Vector x, ESI_Vector y)
{
  ESI_Vector z;

    (composeList != NULL) {
    ESI_Vector_clone(x, &z);
    ESI_Operator_apply(composeList[0], x, z);
    ESI_Operator_apply(composeList[1], z, y);
    ESI_Vector_destroy(z);
  }
      0;
}
```

```
int ESI_Operator_composeLeft(ESI_Operator op, ESI_Operator *newOp)
{
  ESI_Operator_create(newOp);
  (*newOp)->composeList    =    ESI_Operator[2];
  (*newOp)->composeList[0] =    ;
  (*newOp)->composeList[1] = op;
     0;
}
```

```
int ESI_Operator_composeRight(ESI_Operator op, ESI_Operator *newOp)
{
  ESI_Operator_create(newOp);
  (*newOp)->composeList    =    ESI_Operator[2];
  (*newOp)->composeList[0] = op;
  (*newOp)->composeList[1] =    ;
     0;
}
```

■ Splittable Operators

The interface **OperatorSplittable** allows the operator to be factored into F = LR, and further allows the creation of a new operator LFR from any given operator F. Not only does this interface enable black–box split preconditioning, but also optimizations such as Eisenstat's trick.

5. Solver Composition

■ A Motivating Example

We begin with the most elementary type of solver composition, iterative refinement. The idea is that the solver has not done the best job on the first try and we will need to correct the solution it has produced. The system is given by

$$Ax^* = b, \tag{1}$$

where x^* is the true solution, and define the residual r as

$$r = b - Ax_1 . \tag{2}$$

The approximate solution x_1 is given by

$$x_1 = \hat{A}^{-1} b, \tag{3}$$

where we use a hat to indicate an inexact numerical process. Now the idea is to use the residual to define a correction to our approximate solution

$$\Delta x_1 = \hat{A}^{-1} r . \tag{4}$$

We can see that this makes sense by looking at the result of an exact solve in equation (4),

$$A^{-1} r = A^{-1} (b - Ax_1) = x^* - x_1 , \tag{5}$$

so that

$$x_1 + \Delta x = x^* . \tag{6}$$

And, in fact, if the rigorous error analaysis is carried out, it can be shown that the solution actually improves even using an inexact solve. This idea may be extended by allowing the matrix to vary at each step, so that we have

$$\Delta x_k = \tilde{A}^{-1} r_k . \tag{7}$$

where $\tilde{A}$ is another approximation to A. This leads directly to a method for composing multiple solvers (or the same solver with itself) by acting successively on the residual of each previous computation.

Thus we should include in the **Solver** interface a **getResidual** function which returns the current residual vector. We then include **resComposeLeft** and **resComposeRight** functions for solver composition, which operate in an identical fashion to **composeLeft** and **composeRight** from the **Operator** interface. Every solver will now be required to store the current residual of the calculation, but this does not seem to be an undue burden as this is the most common element of stopping criteria. An example implemention of this idea is given below:

```
int ESI_Solver_resComposeRight(ESI_Solver S, ESI_Solver T, ESI_Solver
*ST)
{
  ESI_Solver_create(ST);
  (*ST)->resComposeList    = (ESI_Solver *)
ESIMalloc(2*      (ESI_Solver));
  (*ST)->resComposeList[0] = S;
  (*ST)->resComposeList[1] = T;
        0;
}
```

```
int ESI_Solver_apply(ESI_Solver S, ESI_Vector x, ESI_Vector y)
{
  ESI_Vector r, z;

    (S->resComposeList != NULL) {
    ESI_Vector_clone(x, &z);
    ESI_Solver_apply(S->resComposeList[0], x, y);
    ESI_Solver_getResidual(S->resComposeList[0], &r);
    ESI_Solver_apply(S->resComposeList[1], r, z);
    ESI_Vector_axpy(y, 1.0, z);
    ESI_Vector_destroy(z);
  }
        0;
}
```

■ General Solver Composition

In general, we may put most instances of solver composition in the framework of nonstationary linear iterative methods, for which

$$x^{k+1} = x^k + M_k (f - Ax^k) \equiv x^k + M_k r^k \tag{8}$$

where f is the rhs vector, A is the system matrix, x^k is the current approximation to the solution, and M is an arbitrary matrix which may depend on the iteration index. Thus we may mimic any solver–preconditioner combination in this family. A popular representative would be the two level multiplicative Schwarz algorithm[5], which may be expressed as

Algorithm 5.1: Two Level Multiplicative Schwarz

1. $x \leftarrow R^T A_C^{-1} Rf$
2. For $i = 1 \cdots p$
    3. $x \leftarrow x + B_i (f - A_F x)$

where f is the rhs vector, A_F is the system matrix, A_C its coarse representation on the space defined by the projector R, and the B_i are preconditioners on each domain. All of the variations on this theme given in Smith et. al.[5] may be incorporated using only composition and the addition operation. This may also be extended to full multigrid methods[3], which calculate a series of corrections based on coarser representations of the system operator.

References

1. The ESI Forum is located at http://z.ca.sandia.gov/esi.

2. *Design Patterns: Elements of Reusable Object–Oriented Software.* Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Addison–Wesley, 1994.

3. *Analysis of a Multigrid Method as an Iterative Technique for Solving Linear Systems.* Anne Greenbaum. SIAM Journal on Numerical Analysis, **21**(3), 1984.

4. *Petsc 2.0 Users Manual.* Barry F. Smith, William D. Gropp, Lois Curman McInnes, and Satish Balay. Argonne National Laboratory, TR ANL—95/11, 1995. Available via ftp://www.mcs.anl/pub/petsc/manual.ps

5. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Barry F. Smith, Petter E. Bjørstad, and William D. Gropp. Cambridge University Press, 1996.